Steps **from Customer's idea** to **a working Frontend**

- **A Frontend point of view**   v. 2019-05-07 12:23 by JV

    (Will be updated until week before each exam, and after each exam)

This list contains case-related things to study for the Frontend/React/Redux/Ajax exam.

Rough list of preceeding (or in Scrum: iterated) things happening parallel to front-end programming:

1. Customer has the idea
2. Requirements engineering (Interviews, questionnaires, workshops, old system views and outputs, …)
3. UX design, traditionally iterating incrementally between
    a. Use case (Business Process/Navigation) design
    b. View / layout design
    c. (Conceptual Data model design, that was handled in the other doc)
    d. (Logical etc. level Data model design, that was handled in the other doc)
4. UI Mockups. Even partially working Prototypes, or just Wireframe images

----

Setting up the Frontend project - Node, Node<u>mon</u>, npm/yarn, React, Redux, AJAX with Axios, (style libs and files)

5. installing *create-react-app* and running it to generate a running basic node & react frontend development-time project framework (thus no need to say npm init)
6. using npm or yarn to add more module (redux, axios, material-ui, icons, form stuff…) references to the *package.json*.
7. using *npm install* to install all the modules mentioned in *package.json* even if getting just project source from Git
8. the create-react-app has the static parts of the app in *public* folder, e.g. HTML layout, images
9. to be able to use a common theme for all of your Material-UI React components, wrap the root React component inside a MuiThemeProvider (Mui = Material-UI) that is given your theme file reference
10. to be able to connect your React components to Redux, we need to wrap the (now) React Material-UI root element inside a Provider object and we give it the root reducer which knows how to create initial state of all the reducer initial states when the application starts from nothing.

    Note that the create-react-app generated a framework, which is only the **development-time version**:

    - There is Node.js for starting an auto-refreshing (by nodemon) server at port 3000 or other.
    - Programmers can see a lot of readable files well-organized in several easy-to-browse

    The runnable **production version** will be generated with command **npm build**. The runnable version will:

    - …not have any Node.js or nodemon at all!!! = no development time server running
    - no .jsx anymore
    - All code will be optimized and packed to few static JavaScript/HTML/CSS files and served from a static folder of your production server to the browser JavaScript engine/environment that runs those like there was no development version ever

Testing and running the project – Following <u>windows</u> enlighten us about how/whether the views work

i. Browser window running the app and showing the HTML based on the Views and Components
ii. Browser console (errors, warnings, and our own debugging with console.log or console.dir)
iii. Browser console, networking window   (Press the Split console to get two browser dev tools visible)
iv. Front-end node console (where you run e.g. npm start. Compilation errors etc. in here)
v. Back-end node console (where you run e.g. npm start, npm run db:init. Backend prints and errors here)
vi. Console where you created the SSH pipe. Usually you don't have to keep that console open, but with some consoles the SSH (pipe) process dies when you close the console!
vii. Redux Dev Tools (Right-click and "To left". E.g. select last dispatched Action + see what is in the State. Look carefully and be exact!)
viii. PostMan to check input and output of the needed back-end services
ix. Browser dev tools, tool for checking how page will scale for a mobile device
x. (React dev tools?)(Browser debugger?)
xi. Front and back consoles for finding out which processes are running and which TCP/UDP ports they listen to

Understanding the Front-end part of our architecture

Reducers

- When the application will start in the browser, the reducers will get called (the @@init action) to create the initial state of the Redux store. Reducers use your default values for the state, look at a reducer file for understanding this, in reducers folder.
- Reducers are functions that are passed **actions (=objects)** so that they are able to find out how to modify the **old state** and => return the **new state**.
- Redux calls the reducers when an Action with certain ACTION_TYPE will be dispatch to it
- Redux uses the reducer to update the state in the Redux Store
- (And because the state in the Redux changes, the mapStateToProps causes Redux-connected React components props to change, which in turn causes the component to re-render…)

Actions

- Objects that depict what kind of change we want to happen.
    o Action type, the allowed ones listed in one place in the project, E.g. IDEA_DELETE_REQ
    o (Possibly, but not always) Other payload, like idea id: 3001

Action-related helper functions, e.g. ideaDelete_REQ in our naming convention

- Functions that
    o Set the loading flag up in the state, meaning the state for that part might be changing, as we have an ongoing backend service
    o dispatch the _REQ action indicating the start of the action that will go to backend
    o do the AJAX
    o and setup the call-back for the time when the AJAX action (asynchronous) is ready
    o Then, if successful return, dispatch the _OK action and put the data via reducer to the state
    o or, if unsuccessful, dispatch the _X action to indicate failure

React

Virtual DOM

- React will update just its own virtual DOM, and not the real browser DOM, until it knows all the changes. Thus, e.g. if you Mike was removed and Larry was added, React is able to optimize, not remove nor add any nodes from the browser, but just manipulate browser DOM "Mike"=>"Larry" (The so called 'delta', change), if they by luck would be in the same location in the list. (Not best example, but works with any such update that is optimizable)

Root component

- Includes other components, which in turn will be rendered to the page too.
- Mother components often pass data and event-handler function references to its child components.

Your components

- Presentational components, like e.g. LocationItem,
    o get data in their props from the mother
    o get possibly also the event-handler function references from the mother, map their events to those event-handlers
- Container components, like e.g. LocationList
    o get data from the Redux store (mapStateToProps)
    o dispatch Actions using the action-related helper functions mapped to the props (mapDispatchToProps)
    o connected to Redux with the connect ….   withStyles ….   –way

Think of a mother component that has a child component included inside:

The ways mother can pass anything to the child:

- data via props, methods via props
- cause a change to happen at backend and the child component would be also connected to AJAX/Redux and get the data that way via AJAX/Redux from the backend
- cause a change to happen just in Redux (Rare cases whe no need to change the DB data yet)

The ways child can pass anything to the mother:

- ~~state – No, child doesn't see mother's state~~
- ~~props – No, child doesn't see mother's props~~
- by calling the offered functions provided by Mother and adding the data to the function calls as parameter(s).
- cause a change to happen at the backend and mother component would be also connected to AJAX/Redux and get the data that way via AJAX/Redux from the backend
- cause a change to happen just in Redux (Rare cases whe no need to change the DB data yet)

React **Component State**

- We use mainly Component's **Redux state** connected Props in our case. But we can also use State of the Component object (Not to be confused with the Redux state). See the ProjectList views filtering view where the Search string / criteria is put to Component's state and then used when dispatching the search helper function
- While using the React Component's state we cannot modify it directly by assignment. We should do it with the setState( ) function and pass it an ad hoc -object where the only the to-be-changed properties are, the so called 'delta', the changes.

React Component Lifecycle event-handlers (Simple version to remember)
- When component created to eventually come visible to the screen
    o Component JavaScript object created
    o The possible constructor called, ((And there base class constructor super( ) called as very first))
    o **render( )** - After render the component will come visible
    o **componentDidMount( )** – We put fetching the data often here, so component is often rendered without data first, to not to hang the view
- When Components state or props change (E.g. because Redux store changes, and is mapped to props)
    o render( ) – Updates will come visible
    o componentDidUpdate( )

- When unmounting the Component (E.g. leaving the view where that component is used)
    - **componentWillUnmount( )** - Here we could e.g. dispatch resetting locationCurrent to null

11. In foreign key handling, there are basically three different kind of situations
    a. standalone table that has an id like Category or SkillCategory – No foreign key saved to table
    b. table with and id and foreign key(s) to others, like Idea or Skill
    c. table without an id, PK composite key of mainly foreign keys to other tables, like Comment by a Member for an Idea, or ProjectWorker
12. Possibly some UI features might suggest transactional backend operation, where one user action causes many lines to be updated in the database
13. Remember the ICT principle, the rule of one: Only one place where all colors are from, only one place where server address etc.

SOME SPECIFIC STUFF TO REMEMBER IN React Development

1. Your component names must start with Caps, small letters reserved for HTML

```
class PlayerListItem extends React.Component { .......
```

   // If you would accidentally use small letter, the JSX parser will not know how to work correctly

2. While returning JSX "XML-like" markup from the render method, either wrap the "XML-like" JSX in parentheses ( ), or start it from the very same line as the keyword "return".

3. How to read this JSX?   `<SomeElement attribute={{a:foo}} />`

>   The outer { } takes us from the JSX "XML-like" mode to JavaScript mode. Then the inner {a:foo} would just mean creating an ad-hoc JavaScript object with property called 'a' with its value copied from variable 'foo'.

 3.b Like above, also in `<SomeElement attribute={{a}} />` we would create a JavaScript object, this time with {a} = object with one property 'a' with its value copied from a variable called 'a' in the current context.

4. In many/some places you'll **have to** use the arrow function in call-back function definition. (As arrow function takes the 'this' reference from the outer lexical context = from the React component). A normal function would not bind the "this" to the outer context = component, like we want to happen.

5. The create-react-app sets up the development environment, with another Node.js server than the Node.js, possibly trying to listen to port 3000. (To avoid probs, we could use port 8686 for the development time front end Node.js "server". Back-end Node.js server could use 8787)

6. The final front-end we get when we copy the contents of the /build folder and publish it on some real web server as static .js .html and .css files E.g. on Proto, Myy, AWS, or Heroku Then we won't have front-end Node.js server at all anymore!!! Just static web server is enough. Most likely with port 80. The the execution environment will then be just the browser.

7. Back-end REST API server (E.g. AWS, Heroku, or Proto) will continue to run Node.js., e.g. with that 8787 port. Myy as static content web server won't be fine for Backend, as Myy doesn't allow us run any servers.

*) "XML-like": JSX is not valid XML, as it's not even well-formed XML. It follows some but not all of XML structure ideas.